AGP and 3D Graphics Software

Information for Developers and ISVs

From Intel® Developer Services
www.intel.com/IDS

March 1996

CONTENTS

Summary

The Accelerated Graphics Port (AGP) in Pentium® II processor systems will bring faster and better graphics to the PC. While physically similar to the existing PCI (Peripheral Components Interface) bus, AGP offers new opportunities for performance. Application software, especially 3D graphics, can exploit these opportunities by allocating and managing memory appropriately. However, software that does no AGP optimizations should also benefit. With AGP-targeted applications, users should see more texture, more detail, and higher screen resolution at higher frame rates.

This document should demystify AGP for 3D software developers. It shows how to use AGP and explains the infrastructures in hardware, OS (Operating System), and the API (Application Programming Interface) that support it.

Just Another Bus, Right ?

AGP has two fairly simple facets: (1) the wires, and (2) **DI**rect **M**emory "**E**xecute" (we call it DIME here for pronounce-ability). AGP uses a connector similar to PCI, with 32 bits of multiplexed address and data. While the PCI bus, clocked at 33 MHz supports a maximum of 132 MB/s, AGP at 66 MHz yields a surprising 528 MB/s peak. It gets this quadrupled speed in most implementations, by transferring data on both the rising and falling edges of the 66 MHz clock. Note that some lower-cost hardware may choose to support only one transfer per clock, for 264 MB/s. Of course, throughput will vary among various systems and applications, but usually they obtain about 50-80% of peak values in real world sustainable transfers.

AGP sustains transfers closer to peak values than PCI could, because AGP incorporates a few extra signal wires to enable "pipelining" and queuing of requests, (Figure 1) it overlaps the memory or bus access times for request "n" with the issuing of requests "n+1" and "n+2", etc... In the PCI bus, request "n+1" does not begin until the data transfer of request "n" finishes. While both AGP and PCI can "burst" (transfer multiple data items continuously in response to a single request), that bursting only partly ameliorates the non-pipelined nature of PCI. The depth of AGP pipelining is implementation dependent, and transparent to application software. Furthermore, AGP contains 8 extra "sideband" address lines which allow the graphics chip to issue new addresses and requests simultaneously with the continuing movement of data from previous requests on the main 32 data/address wires. In Figure 1, the sideband simultaneous transfers would allow address "A_{n+1}" to be issued simultaneous with data transfer D1, rather than after D6.

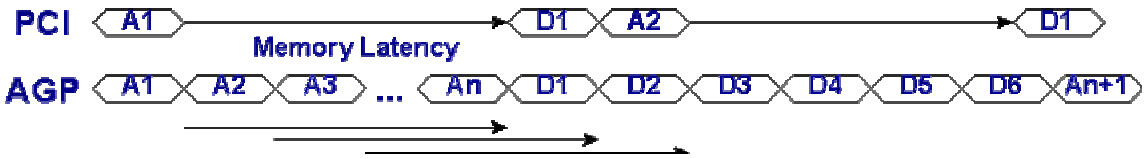

Figure 1: Non-pipelined PCI vs. AGP A_n is the address of the request, and D_n is the result

To permit the high clock rate, AGP really isn't a "bus" with multiple seats like PCI; it's more like a sports car, allowing only two devices (chip set and graphics chip). Figure 2 shows a simplified diagram of an AGP system, with peak bus bandwidths and memory usage. The unnamed bus on the other side of the chip set, between the memory controller and system memory is typically 64 bits wide, and must be shared by CPU, graphics chip, and PCI bus accesses to system RAM. Because of the larger amount of RAM wired to it, its clock rate is usually slower than the graphics local memory bus. The initial chip sets will support 528 MB/s peak on that system bus, but future versions may increase it. The local memory 800 MB/s peak comes from a 64-bit bus at 100 MHz.

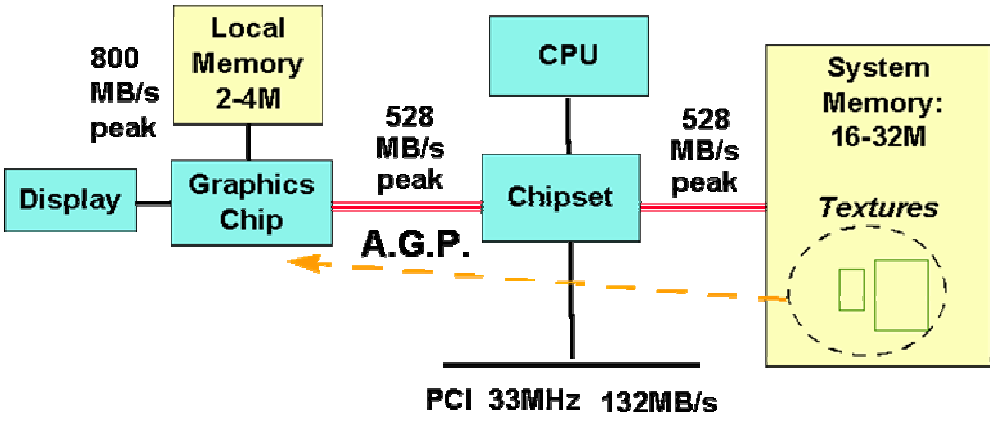

Figure 2: Typical AGP System Block Diagram

So-called "AGP memory" is just dynamically-allocated areas of system DRAM, which the graphics chip can access quickly (shown encircled in dotted lines in Figure 2). The quickness comes from built-in hardware in the chip set which translates addresses, so that the graphics controller and its software can see a contiguous space in main memory, when in fact the pages are disjointed. Thus the graphics chip can access large data structures like texture bitmaps (typically 1KB to 128KB) as a single entity. The built-in chip set hardware is called the GART (Graphics Address Remapping Table), similar in function to the paging hardware in the CPU chip. See Figure 3 for an illustration of the address mapping.

The processor "linear" virtual addresses get translated by its paging hardware into physical addresses. These physical addresses are used to access system RAM, local Frame Buffer, and AGP RAM. The CPU accesses to the Local Frame buffer and AGP RAM use the same addresses as the graphics chip does; for that reason, the operating system sets up the CPU paging hardware to a straight 1:1 non-translation of virtual to physical address.

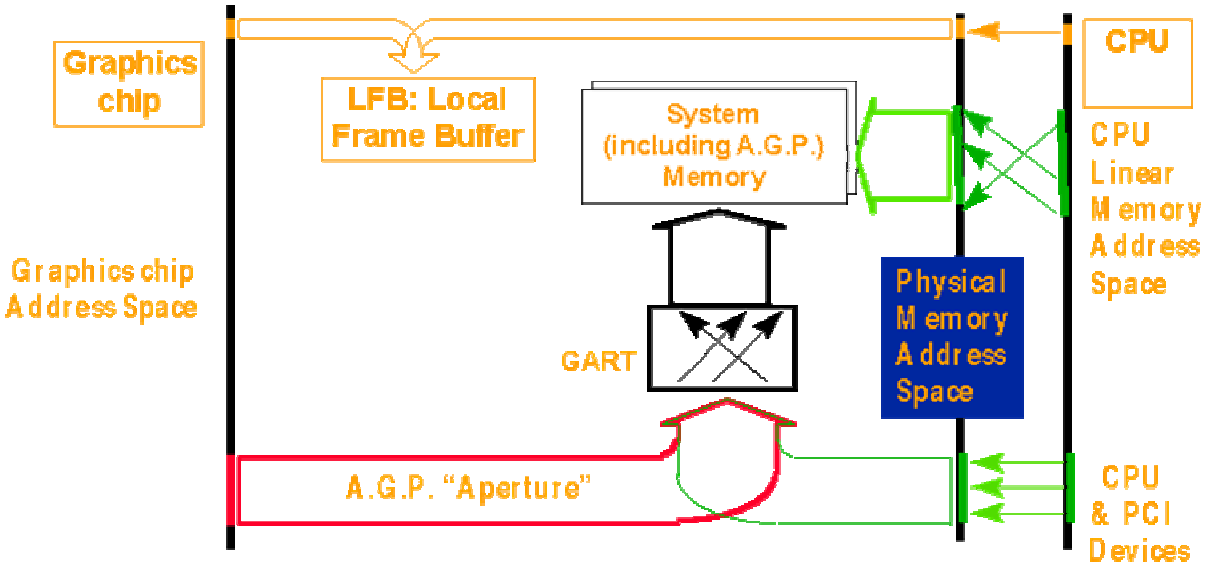

Figure 3: AGP Address Mapping

For accesses to AGP RAM, the graphics chip and CPU use a contiguous "aperture" of several megabytes. But the GART translates these to various, possibly disjointed, 4KB page addresses in system memory. PCI devices access to the AGP memory aperture (for example, for live video capture) also go through the GART. Note that in future driver and chip set implementations, CPU accesses may not need to use the GART, as the CPU page tables may be configured to do the same remapping. But such a change should be transparent to applications software. As you might guess, this GART must be initialized and updated by the operating system software, as 4KB pages are allocated or de-allocated from the AGP address aperture.

Direct Memory Execute (DIME)

All pixels drawn in 3D graphics scenes are either texture-mapped, or "shaded." Shading simply means assigning the same color to every pixel in a region, or smoothly varying the color from one pixel to the next. This requires no bitmap fetches to create the pixel. But texture-mapping consists of fetching one, two, four, or eight texels (**tex**ture **el**ements) from a bitmap, averaging them together based on some mathematical approximation of the location in the bitmap (or multiple bitmaps) needed on the final image, and writing the resultant pixel to the frame buffer. The texel coordinates are non-trivial functions of the 3D viewpoint and the geometry of the object onto which the bitmap is being projected.

Fully-implemented AGP graphics chips can exploit main memory directly (DIME, **DI**rect **M**emory **E**xecute) for the complex operation of texture-mapping. Without a DIME, most PCI-generation graphics controllers can only texture from local graphics memory attached directly to the chip, (Figure 4). Initially, some "first generation" AGP graphics chips will do only PCI-like bus mastering, or maybe none at all, although they will benefit from the faster transfer rates of the new bus. But they will have to transfer the entire texture map to their local memory via AGP, then re-access the texture from that local RAM to get individual texels for mapping.

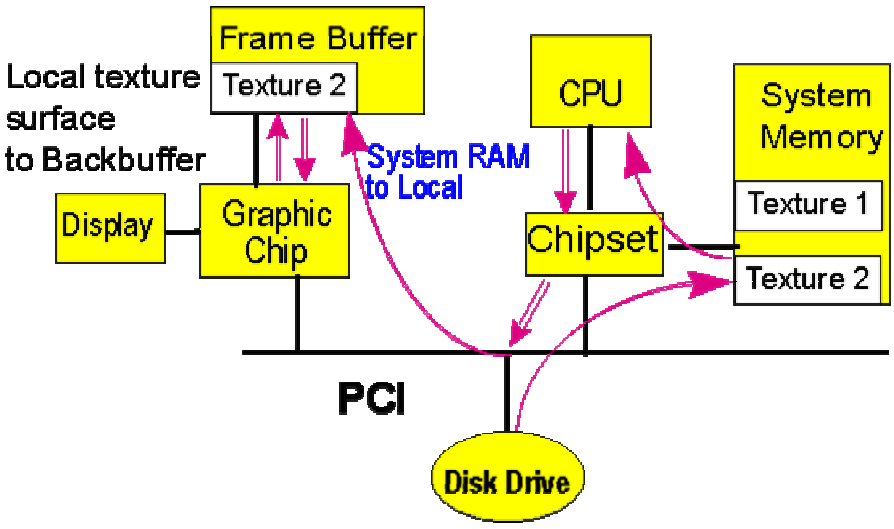

Figure 4: Texture Data Flow with PCI requiring an extra copy in Frame Buffer RAM

If the chips do not bus-master, then the CPU must fetch the texture from main memory and write it via the bus to the frame buffer. Thus, redundant copies of the textures exist -- one in local and another in system RAM. However, as Figure 5 shows, the DIME implementation avoids the redundant copies and avoids extra data movement.

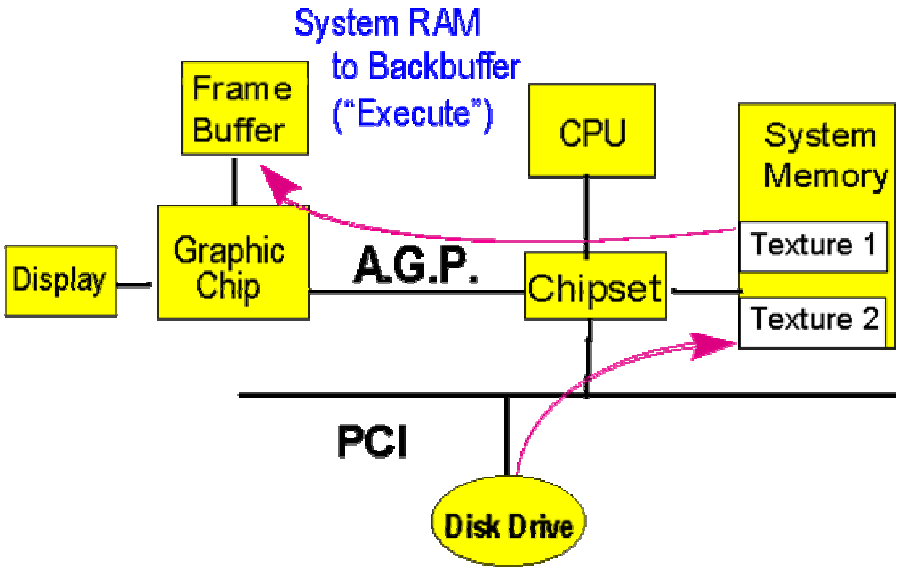

Figure 5: Texture Data Flow with AGP Direct Memory Execute

Graphics local RAM is usually more expensive than generalized system memory and it cannot be used for other purposes by the OS when unneeded by the graphics of the running applications. The graphics chip needs fast access to local memory for screen refresh, Z-buffers, and pixels (front and back-buffers). For these reasons, programmers can always expect to have more texture memory available via AGP system memory. Keeping textures out of the frame buffer allows larger screen resolution, or permits Z-buffering for a given large screen size. Most applications could use 2-16 MB for texture storage. By using AGP and DIME, they can get it.

In summary, the benefits of AGP include:

1. **Peak Bandwidth** 4x the PCI bus, and higher sustained rates via sideband and pipelining.

2. **Direct Memory Execute** of textures.

3. **Reduced Contention** with the CPU and I/O devices for bus and memory access. The PCI bus serves disk controllers, LAN chips, and possibly video capture. AGP operates concurrently with, and independent from, most transactions on PCI. Further, CPU accesses to system RAM can proceed concurrently with the graphics chip's AGP RAM reads, because of so-called out-of-order and queuing hardware support in the chip set. So in spite of heavy access from the graphics chip, there should be no audio breakup or other CPU degradation.

4. **An "extra port" to the graphics chip** for memory access, so it can concurrently read textures from AGP memory while reading/writing Z-values and pixels from local memory. Using the bandwidths from Figure 2, the graphics chip gets 1.3 GB/s peak by using both ports simultaneously, versus "only" 0.8 GB/s from the local RAM.

5. **Allowing the CPU to write directly to shared system AGP memory** when it needs to provide graphics data, such as commands or animated textures. Generally the CPU can more quickly access main memory than it can graphics local memory via AGP, and certainly faster than via the PCI bus.

So what should an application software developer do about AGP ? There are two possibilities: Do nothing, or optimize for AGP For both cases, the big benefit of AGP is MORE, LARGER textures for 3D graphics realism without loss of real-time performance. Today's apps usually must limit themselves to less than 2MB of textures at any time with graphics hardware accelerators. AGP will change that, assuming the application includes the scalability of high-end texture content.

Doing Nothing New

Many existing applications as well as new applications written without special efforts for AGP will run faster or better on AGP systems.

How 3D applications currently manage texture memory

Applications decide which textures should be stored in local memory verses system memory, via a caching or "swapping" algorithm. Typically, applications dedicate a portion of offscreen local memory as frame-to-frame texture swapping space, while the remaining offscreen memory contains commonly used textures (fixed texture memory) - for example, clouds and sea in a flight simulator. Different sizes or numbers of textures are kept locally for 1M, 2M, and 4M graphics cards.

If the hardware can only texture from local memory, the algorithm usually makes great effort to prefetch the needed textures for each frame or scene into local memory. Without prefetching, users will see a noticeable pause in the scene as the software stops drawing while the needed texture is swapped into graphics memory, or even worse, from disk to system memory to graphics local memory. Often even more delay in initial texture loading occurs due to necessary reformatting of textures into a hardware-specific compressed format. See the left side of Figure 6 for a flowchart of this swapping behavior.

Applications may reserve part of the local RAM for swapping, and leave part of it permanently loaded with "fixed" commonly used textures. Depending on the number of textures per frame, the algorithm may vary the proportion of memory allocated for texture swapping and fixed texture memory. Scenes which contain a large number of textures tend to have less texture reuse; these benefit from larger texture swapping space.

DOS Applications

Of course Direct Memory Execution of textures requires the GART, because of the virtual addressing scheme used in today's operating systems. But for applications running under yesterday's operating systems (E.G., DOS) without virtual addressing, the GART serves no purpose. Old applications running under DOS will see the benefit of faster AGP speed, but would require some driver work to turn on the DIME features of the chips. It is unlikely that work will happen.

Windows Applications

Unmodified Windows* applications can benefit from AGP, because the OS and DirectDraw* have changed slightly to support it by default. Below appears the text from Microsoft's website:

> "Microsoft will support AGP hardware with enhancements to both Windows* *95 [Ed: actually, the version code-named Memphis]* and Windows NT* *[Ed: NT version 5.0]*. The AGP functionality will be exposed to applications via the Microsoft DirectDraw* API. Software support for AGP will be available on future releases of both operating systems. Support for AGP is built into the version of DirectDraw planned for release in the second half of 1997.
>
> Microsoft will provide code in both Windows 95 and Windows NT that will enumerate and initialize devices on the AGP. The operating system will also be responsible for initializing the AGP hardware and managing the mapping from system memory to the video hardware address space. This **support is mostly behind the scenes** and will not be directly accessible to applications programmers. Applications that want to take advantage of the AGP capabilities can do so using the DirectDraw API.

> DirectDraw will receive information from the display driver about the amount of AGP memory that the display hardware is capable of addressing. An application can request access to this memory by allocating surfaces through the DirectDraw API. DirectDraw completes this request by using underlying system services to commit and pagelock system memory pages and then set up the AGP mapping table to allow the display hardware access to these pages.
>
> If the operating system determines that these pages cannot be locked without seriously impacting system performance, the surface allocation request will fail. In some cases, the operating system might request DirectDraw to free up AGP memory that has already been allocated. DirectDraw will invalidate the necessary surfaces and unlock the affected memory pages. This is very similar to the situation that occurs when a display mode change takes place and display memory must be reconfigured. DirectDraw applications already deal with the possibility that a video memory surface might be lost at any time." (http://www.microsoft.com/hwdev/devdes/msagp.htm Feb 1997).

Note that the OS *locks* the pages it allocates to AGP, so that they cannot be swapped out to disk. Currently (as of Q2'97) it determines the amount of available memory allocable to AGP via a key in the system registry, but we expect that that amount will be approximately equal to total memory minus the MBytes needed by the OS and currently running applications. For example, in a 32M system, probably 16M to 24M will be AGP-able.

For current hardware implementations, the OS will make AGP memory (like other video memory) non-cacheable, so that there is no coherency problem between the CPU caches and the data that the graphics controller uses. Otherwise, graphics controller accesses to AGP RAM would require "snooping" the CPU caches, which would cause delays in execution in some cases. CPU reads from uncached RAM are slow, so algorithms should avoid CPU reads from AGP main memory as well as from graphics memory.

Note that in Pentium® II processor systems, this non-cached graphics memory will be marked by the OS as "Write Combining" (WC), which gets significantly faster CPU write-access than straight "Uncacheable" (UC). WC memory areas let the CPU "combine" multiple discreet writes into a burst-write on the bus when the bus is available, using dedicated write-buffers built-in to the chip. Except for the faster speed, WC should remain transparent to applications. While the CPU read-access speed is no faster for WC than UC, the use of UC memory will cause Pentium II processors to serialize execution, which will probably slow the execution significantly. The fact that multiple writes can get combined and smashed together before getting outside the CPU chip can have some impact on hardware device drivers, which may depend on multiple sequential writes to the same location, and "strong ordering" of memory writes.

Default DirectDraw Memory Allocations

Unless the application specifically requests otherwise, Microsoft DirectDraw will by default assign surfaces (I.E., allocate memory) in the following order:

1. Local graphics controller memory (called `LocalVidMem` or Frame Buffer or Offscreen RAM)
2. AGP main memory (also called `NonLocalVidMem`)
3. System memory

DirectDraw allocates RAM if there is enough space in the corresponding area for the request from the application. Say for example, an application wants to use a 640x480 window @ 16bits/pixel, with double buffering and a Z-buffer. Those three buffers take 600K each. This example application also wants to use 64 textures, each 256x256 @ 16 bits/texel. Those textures add up to 64*128K = 8M total. Assume the hardware includes 2M of LFB (Local Frame Buffer) memory.

The application first requests the frontbuffer (600K), backbuffer (600K), and Z-buffer (600K), which DirectDraw puts in local graphics memory. As Table 1 shows, only 200K remains in Local memory. So DirectDraw will put one texture into that space, and the other 63 into AGP memory. If the graphics hardware had 4M of Local RAM, then 17 textures fit in the leftover 2.2M there, and the remaining textures in AGP memory.

Sample RAM allocation, for 16-bit pixels & Z, with 8M of textures desired.	**640x480 Window and 2M Local Video RAM**	**800x600 Window, 4M Local RAM**	**1024x768 Window, 4M Local RAM**
Front buffer	600K	0.92M	1.5M
Back buffer	600K	0.92M	1.5M
Z-buffer	600K	0.92M	Won't fit
Remaining Local RAM (for textures, scratch, cursors)	200K (from a 2M FB)	1M from a 4M Frame buffer	1M from a 4M Frame buffer
AGP used for textures	7.8M	7M	8M

Table 1: RAM Allocation Examples

We have assumed 16-bit pixels, 16-bit Z, and 16-bit texels because 16-bits is commonly supported by 3D hardware accelerators, and yields good visual quality. Some applications will use 8-bit textures and avoid using a Z-buffer in order to free more local memory for textures. For software-only unaccelerated implementations, 8-bit pixels are typically used and no Z-buffer. Thus more textures can fit in local memory. Ironically, SW-only implementations do not want or need textures in local memory. The SW algorithm needs quick CPU access to textures and thus locates them in system memory (and the CPU caches).

If the application tried to allocate a huge amount of texture surface, say 24M in a system where the OS allows only 16M of AGP memory, then the last 8M of the requested 24M would be either refused or allocated in system memory. The application must check the return code from the `CreateSurface()` call to determine success of its allocations, as explained in a later section.

What if the graphics chip cannot texture from AGP memory ? Luckily, this case prevents DirectDraw from allocating any NonLocalVidMem for texturing. The graphics chip driver reports its capabilities to the OS and DirectDraw, and if the chip cannot DIME, then DirectDraw will allocate only Local video memory and system memory to the application. The DirectDraw flag `DDSCAPS_NONLOCALVIDMEM` tells the application whether it received AGP RAM, as discussed in a later section. Similarly, if the graphics chip cannot texture from local video memory, DirectDraw will not allocate any texture surfaces locally.

Targeting AGP

True AGP-compliant hardware with DIME can actually make applications simpler. But PC hardware for AGP will come in three flavors, and software probably will want to support all three:

1. **Without a DIME**: this hardware sits on AGP, but does not exploit its DIME features. It just transfers data faster than a PCI device could. It probably does not exploit the pipelining capability or sideband addressing.

2. **DIME** (Direct Memory Execute): this hardware renders textures from system (AGP) memory. Thus the application does not need to swap textures into local memory. The hardware may or may not be able to texture from local memory also. It may perform faster when NOT texturing from local, due to conflicts for access to local memory for pixel writes, screen refresh, texel reads, and Z-values.

3. **DIMEL** (Direct Memory Execute and Local also): this hardware runs best when concurrently exploiting both local memory and AGP memory for texturing. Frequently-used textures or smaller textures would best reside in local RAM, while larger less-frequently used textures reside in system memory. Thus the bandwidth drain on main memory is minimized, reducing conflicts between the CPU and graphics chip.

To target DIME hardware, the application follows the flowchart on the right side of Figure 6. It allocates space for each texture in AGP memory, then loads there the textures it will need for this frame or the entire scene. It does not bother with the step shown in dotted lines, namely, copying some textures to the local RAM.

If it could not fit all textures into the AGP memory which DirectDraw agreed to allocate, then the app must eventually copy some more textures from the disk into AGP RAM. Very realistic flight simulators or other applications using large amounts of textures may need to stream textures from disk or network into AGP RAM, no matter how much RAM DirectDraw gave them.

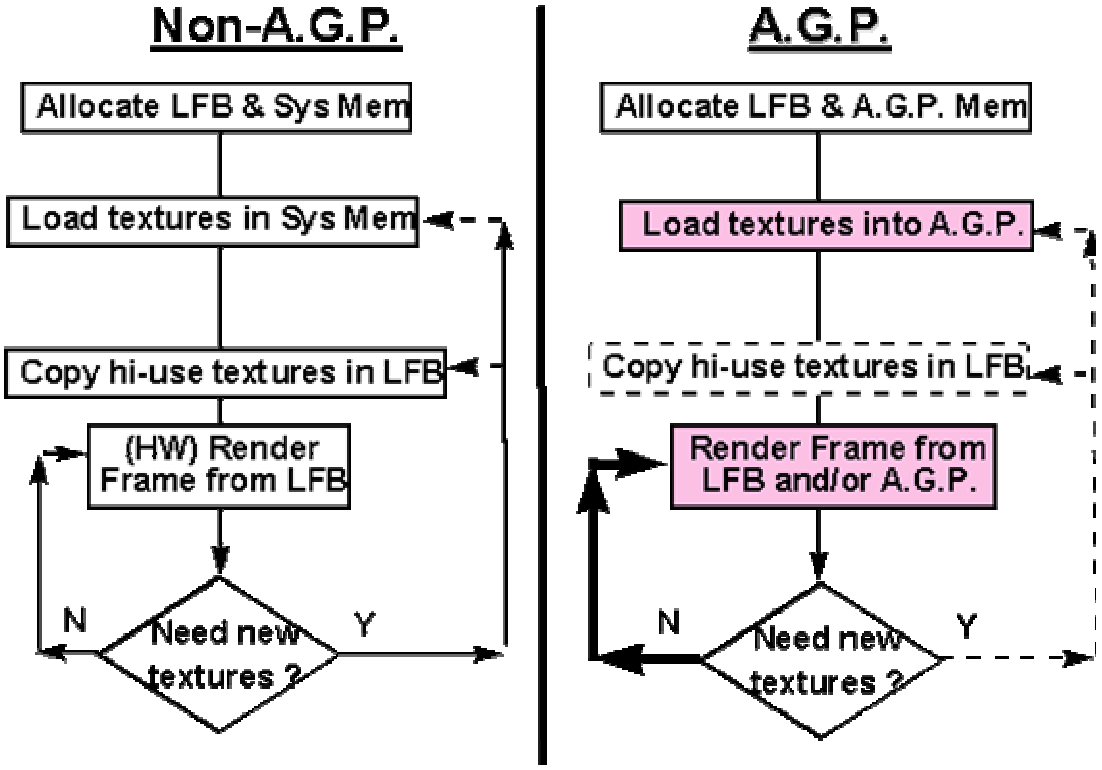

Figure 6: Texture Memory Management, before and after AGP

The application may benefit from using MIP-maps with AGP, as MIP-maps (pre-filtered multi-resolution texture maps) tend to increase the "locality" of memory access during texturing. That is, the lower-resolution version fits into a small area of system RAM, and as the graphics chip puts the texture on an object far from the viewpoint, it accesses that subsampled version of texture all within a small memory region. Without MIP-mapping, the chip must skip over many bytes of the single-resolution larger texture to find the right texel for each pixel - so memory addresses jump in large increments, and the RAM bandwidth is lowered.

The app can request AGP memory, by using the following DirectDraw code:

```
//Ask for a simple offscreen surface in AGP RAM

ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | DDSCAPS_VIDEOMEMORY |
DDSCAPS_NONLOCALVIDMEM;

dwHeight = 256; dwWidth = 256; //256x256, for example

ddrval = lpdd->lpVtbl->CreateSurface(lpDD, &ddsd, &lpOffScreen, NULL);

if( ddrval != DD_OK ) { // surface was not created

return FALSE;} ;
```

The app can check whether it is accessing an AGP surface, using the following code:

```
// Find out if a surface is in AGP RAM
lpOffScreen->lpVtbl->GetSurfaceDesc(lpOffScreen, &ddsd);
```

```
if(ddsd.ddsCaps.dwCaps & DDSCAPS_VIDEOMEMORY)
{
if( ddsd.ddsCaps.dwCaps & DDSCAPS_LOCALVIDMEM )
printf("Surface in local video memory\n");
else printf("Surface in non-local video memory\n"); } //This is AGP
else printf("Surface in system memory\n");
```

Of course a real application would do something more clever than a "`printf()`" upon discovering the memory type. For example, if it was indeed NONLOCALVIDMEM, it would load a texture into the surface. If not it would make a new request for that memory.

Mixing Something Old, Something New

To target DIMEL hardware, which can texture from both local and nonlocal video memory, the application would follow the full set of actions in the right side of Figure 6. It would copy some textures into local memory, until it ran out of local space. Actually that algorithm is the general case required to cover both non-DIME and DIME hardware. If the texture space allocated in main memory by DirectDraw is not AGP, then the application needs to follow the same swapping policy that it followed for PCI graphics controllers. Luckily the swapping should happen faster over AGP.

Sticky Details

Several details remain to be considered, including querying the quantity of AGP memory and PIO (Programmed I/O, where the CPU moves all data) vs. DMA. The application may want to query the total amount of AGP memory available in the system, and the amount already allocated. The `IDirectDraw3::GetAvailableVidMem()` function, with `DDSCAPS= DDSCAPS_NONLOCALVIDMEM`, should return allocatable AGP quantities.

PIO vs. DMA

To get data (for example, decompressed video or animated textures) directly into local graphics RAM, should an application write it directly via PIO (Programmed Input/Output), or should it write to AGP `NonLocalVidMem`, and let the graphics controller then copy it ? Probably the optimal choice will vary from graphics chip to graphics chip, driver to driver, and system to system. Software developers should benchmark the two methods on the system configurations they care about.

By writing to local RAM over the AGP , the CPU can get data to the frame buffer quickest, but the graphics chip must sit idle while the CPU monopolizes its local RAM and/or input FIFOs. Depending on the latency in getting access to the local RAM, and the number of "wait states" imposed by direct writes, and whether the memory is allocated as WC or uncacheable, the CPU will probably spend more cycles writing there than it would writing to main AGP memory. In the case of PCI, the CPU bandwidth writing to the frame buffer has been significantly lower than writing to main memory.

But if DMA is used, the bandwidth of the main memory system is doubly taxed - once for the CPU write, and once for the graphics chip read. And memory space is "wasted" with data that really was needed only in local graphics RAM. Finally, the system memory controller must waste some cycles arbitrating between CPU access and graphics chip access.

Summary and Recommendations

Like the transition from the 16-bit ISA bus to the 32-bit PCI bus in the early 1990s, the transition to the 528 MB/s AGP in the late 1990s should offer a quantum-leap improvement in PC graphics performance, especially for 3D. Existing applications may benefit from the faster bus and the "Direct Memory Execute" of textures. But new applications written to exploit AGP should see noticeably speedier 3D with far richer texture content Application developers should profile the speed of the hardware they target, and choose Direct Memory Execute or local texture caching based on their performance on the particular application. AGP removes the texture size and bandwidth constraints which have previously stifled development of high-quality realtime 3D.

Bibliography

AGP Implementors' Forum, http://www.agpforum.org/ contains the detailed AGP Port Specification and other technical data.

The Intel AGP Web page, /pc-supp/platform/agfxport/

The Microsoft AGP Web page, http://www.microsoft.com/hwdev/devdes/msagp.htm

"Who's Laughing Now", in Boot Magazine, March 1997, pg. 33 reports on ATI's AGP chip.